

# LIFELONG PERCEPTUAL PROGRAMMING BY EXAMPLE

Alexander L. Gaunt, Marc Brockschmidt, Nate Kushman, Daniel Tarlow

Microsoft Research

## ABSTRACT

We introduce and develop solutions for the problem of *Lifelong Perceptual Programming By Example (LPPBE)*. The problem is to induce a series of programs that require understanding perceptual data like images or text. LPPBE systems learn from weak supervision (input-output examples) and incrementally construct a shared library of components that grows and improves as more tasks are solved. Methodologically, we extend differentiable interpreters to operate on perceptual data and to share components across tasks. Empirically we show that this leads to a lifelong learning system that transfers knowledge to new tasks more effectively than baselines, and the performance on earlier tasks continues to improve even as the system learns on new, different tasks.

## 1 INTRODUCTION

A goal of artificial intelligence is to build a single large neural network model that can be trained in a *lifelong learning* setting; i.e., on a sequence of diverse tasks over a long period of time, and gain cumulative knowledge about different domains as it is presented with new tasks. The hope is that such systems will learn more accurately and from less data than existing systems, and that they will exhibit more flexible intelligence. However, despite some work showing promise towards multitask learning (training on many tasks at once) and transfer learning (using source tasks to improve learning in a later target task) (Caruana, 1997; Luong et al., 2015; Parisotto et al., 2015; Rusu et al., 2016), most successes of neural networks today come from training a single network on a single task, indicating that this goal is highly challenging to achieve.

We argue for two properties that such systems should have in addition to the ability to learn from a sequence of diverse tasks. First is the ability to learn from weak supervision. Gathering high-quality labeled datasets is expensive, and this effort is multiplied if all tasks require strong labelling. In this work, we focus on weak supervision in the form of pairs of input-output examples that come from executing simple programs with no labelling of intermediate states. Second is the ability to distill knowledge into subcomponents that can be shared across tasks. If we can learn models where the knowledge about shared subcomponents is disentangled from task-specific knowledge, then the sharing of knowledge across tasks will likely be more effective. Further, by isolating shared subcomponents, we expect that we could develop systems that exhibit reverse transfer (i.e., performance on earlier tasks automatically improves by improving the understanding of the object class in later tasks).

A key challenge in achieving these goals with neural models is the difficulty in interpreting weights inside a trained network. Most notably, subcomponents of knowledge gained after training on one task cannot be easily transferred to related tasks. Conversely, traditional computer programs naturally structure solutions to diverse problems in an interpretable, modular form allowing (1) re-use of subroutines in solutions to new tasks and (2) modification or error correction by humans. Inspired by this fact, we develop end-to-end trainable models that structure their solutions as a library of functions, some of which are represented as source code, and some of which are neural networks.

Methodologically, we start from recent work on programming by example (PBE) with differentiable interpreters, which shows that it is possible to use gradient descent to induce source code operating on basic data types (e.g. integers) from input-output examples (Gaunt et al., 2016; Riedel et al., 2016; Bunel et al., 2016). In this work we combine these differentiable interpreters with neural network classifiers in an end-to-end trainable system that learns programs that manipulate perceptual




---

```

T = 5; tape_length = 4; max_int = tape_length
@Runtime([max_int, 2], max_int)
def add(a, b): return (a + b) % max_int

@Runtime([tape_length], tape_length)
def inc(a): return (a + 1) % tape_length

tape = Input(2)[tape_length]
instr = Param(2)[T]
count = Var(max_int)[T + 1]
pos = Var(tape_length)[T + 1]

pos[0].set_to(0)
count[0].set_to(0)

for t in range(T):
    if instr[t] == 0: # MOVE
        pos[t + 1] = inc(pos[t])
        count[t + 1].set_to(count[t])
    elif instr[t] == 1: # READ
        pos[t + 1].set_to(pos[t])
        with pos[t] as p:
            count[t + 1].set_to(
                add(count[t], tape[p]))

final_count = Output(max_int)
final_count.set_to(count[T - 1])

```

---




---

```

T = 5; tape_length = 4; max_int = tape_length
@Runtime([max_int, 2], max_int)
def add(a, b): return (a + b) % max_int

@Runtime([tape_length], tape_length)
def inc(p): return (p + 1) % tape_length

@Learn([Tensor(28,28)], 2, hid_sizes=[256,256])
def is_dinosaur(image): pass

tape = InputTensor(28,28)[tape_length]
instr = Param(2)[T]
count = Var(max_int)[T + 1]
pos = Var(tape_length)[T + 1]
tmp = Var(2)[T + 1]

pos[0].set_to(0)
count[0].set_to(0)

for t in range(T):
    if instr[t] == 0: # MOVE
        pos[t + 1] = inc(pos[t])
        count[t + 1].set_to(count[t])
    elif instr[t] == 1: # READ
        pos[t + 1].set_to(pos[t])
        with pos[t] as p:
            tmp[t].set_to(is_dinosaur(tape[p]))
            count[t + 1].set_to(
                add(count[t], tmp[p]))

final_count = Output(max_int)
final_count.set_to(count[T - 1])

```

---

Figure 1: (NEURAL) TERPRET programs for counting symbols on a tape, with input-output examples. Both programs describe an interpreter with instructions to MOVE on the tape and READ the tape according to source code parametrised by `instr`. (left) A TERPRET program that counts '1's. (right) A NEURAL TERPRET program that additionally learns a classifier `is_dinosaur`.

data. In addition, we make our interpreter modular, which allows *lifelong learning* on a sequence of related tasks: rather than inducing one fresh program per task, the system is able to incrementally build a library of (neural) functions that are shared across task-specific programs. To encapsulate the challenges embodied in this problem formulation, we name the problem *Lifelong Perceptual Programming By Example (LPPBE)*. Our extension of differentiable interpreters that allows perceptual data types, neural network function definitions, and lifelong learning is called NEURAL TERPRET (NTPT).

Empirically, we show that a NTPT-based model learns to perform a sequence of tasks based on images of digits and mathematical operators. In early tasks, the model learns the concepts of digits and mathematical operators from a variety of weak supervision, then in a later task it learns to compute the results of variable-length mathematical expressions. The approach is resilient to catastrophic forgetting (McCloskey & Cohen, 1989; Ratcliff, 1990); on the contrary, results show that performance continues to improve on earlier tasks even when only training on later tasks. In total, the result is a method that can gather knowledge from a variety of weak supervision, distill it into a cumulative, re-usable library, and use the library within induced algorithms to exhibit strong generalization.

## 2 PERCEPTUAL PROGRAMMING BY EXAMPLE

We briefly review the TERPRET language (Gaunt et al., 2016) for constructing differentiable interpreters. To address LPPBE, we develop NEURAL TERPRET, an extension to support lifelong learning, perceptual data types, and neural network classifiers. We also define our tasks.

### 2.1 TERPRET

TERPRET programs describe differentiable interpreters by defining the relationship between Inputs and Outputs via a set of inferrable `Params` that define an executable program and `Vars` that store intermediate results. TERPRET requires all of these variables to be finite integers. To learn using gradient descent, the model is made differentiable by a compilation step that lifts the

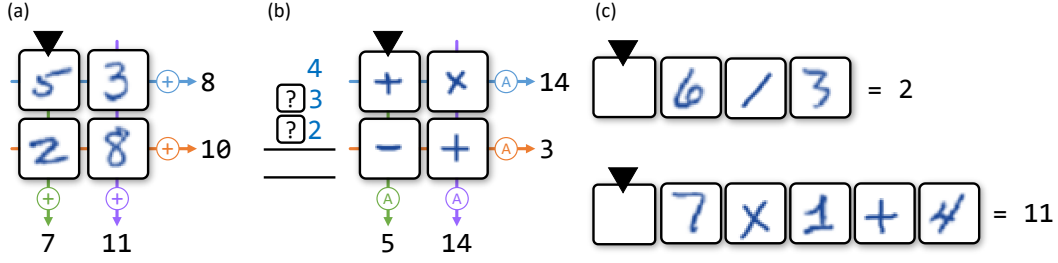


Figure 2: Overview of tasks in the (a) ADD2X2, (b) APPLY2X2 and (c) MATH scenarios. ‘A’ denotes the APPLY operator which replaces the ? tiles with the selected operators and executes the sum. We show two MATH examples of different length.

relationships between integers specified by the TERPRET code to relationships between marginal distributions over integers in finite ranges. There are two key operations in this compilation process:

- **Function application.** The statement `z.set_to(foo(x, y))` is translated into  $\mu_i^z = \sum_{j,k} I_{ijk} \mu_j^x \mu_k^y$  where  $\mu^a$  represents the marginal distribution for the variable  $a$  and  $I$  is an indicator tensor  $\mathbb{1}[i = \text{foo}(j, k)]$ . This approach extends to all functions mapping any number of integer arguments to an integer output.
- **Conditional statements** The statements `if x == 0: z.set_to(a); elif x == 1: z.set_to(b)` are translated to  $\mu^z = \mu_0^x \mu^a + \mu_1^x \mu^b$ . More complex statements follow a similar pattern, with details given in Gaunt et al. (2016).

This compilation process yields a TensorFlow Abadi et al. (2016) computation graph containing many of these two operations, which can then be trained using standard methods.

## 2.2 NEURAL TERPRET

To handle perceptual data, we relax the restriction that all variables need to be finite integers. We introduce a new *tensor* type whose dimensions are fixed at declaration, and which is suitable to store perceptual data. Additionally, we introduce *learnable functions* that can process vector variables. A learnable function is declared using `@Learn([d1, ..., dD], dout, hid_sizes=[ℓ1, ..., ℓL])`, where the first component specifies the dimensions  $d_1, \dots, d_D$  of the inputs (which can be finite integers or tensors) and the second the dimension of the output. NTPT compiles such functions into a fully-connected feed-forward neural network whose layout can be controlled by the `hid_sizes` component, which specifies the number of layers and neurons in each layer. The inputs of the function are simply concatenated. Vector output is generated by learning a mapping from the last hidden layer, and finite integer output is generated by a softmax layer generating a distribution over integers up to the declared bound. Learnable parameters for the generated network are shared across every use in the NTPT program, and as they naturally fit into the computation graph for the remaining TERPRET program, can be trained the same way.

A simple TERPRET program counting bits on a tape, and a related NTPT program that counts up images of a particular class on a tape are displayed in Fig. 1.

## 2.3 TASKS

To demonstrate the benefits of our approach for combining neural networks with program-like architecture, we consider three toy scenarios consisting of several related tasks depicted in Fig. 2.

**ADD2X2 scenario:** The first scenario in Fig. 2(a) uses of a  $2 \times 2$  grid of MNIST digits. We set 4 tasks based on this grid: compute the sum of the digits in the (1) top row, (2) left column, (3) bottom row, (4) right column. All tasks require classification of MNIST digits, but need different programs to compute the result. As training examples, we supply *only* a grid and the resulting sum. Thus, we *never* directly label an MNIST digit with its class.

**APPLY2X2 scenario:** The second scenario in Fig. 2(b) presents a  $2 \times 2$  grid of of handwritten arithmetic operators. Providing three auxiliary random integers  $d_1, d_2, d_3$ , we again set 4 tasks

<pre>(a) # initialization: R0 = READ # program: R1 = MOVE_EAST R2 = MOVE_SOUTH R3 = SUM(R0, R1) R4 = NOOP return R3</pre>	<pre>(b) # initialization: R0 = InputInt[0] R1 = InputInt[1] R2 = InputInt[2] R3 = READ # program: R4 = MOVE_EAST R5 = MOVE_SOUTH R6 = APPLY(R0, R1, R4) R7 = APPLY(R6, R2, R5) return R7</pre>
---	---

Figure 3: Example solutions for the tasks on the right columns of the (a) ADD2X2 and (b) APPLY2X2 scenarios. The read head is initialized READING the top left cell and any auxiliary InputInts are loaded into memory. Instructions and arguments shown in black must be learned.

based on this grid, namely to evaluate the expression<sup>1</sup>  $d_1 \text{ op}_1 d_2 \text{ op}_2 d_3$  where  $(\text{op}_1, \text{op}_2)$  are the operators represented in the (1) top row, (2) left column, (3) bottom row, (4) right column. In comparison to the first scenario, the dataset of operators is relatively small and consistent<sup>2</sup>, making the perceptual task of classifying operators considerably easier. However, the algorithmic part is more difficult, requiring non-linear operations on the supplied integers.

**MATH scenario:** The final task in Fig. 2(c) requires combination of the knowledge gained from the weakly labelled data in the first two scenarios to execute a handwritten arithmetic expression.

### 3 MODELS

We design one NTPT model for each of the three scenarios outlined above. Knowledge transfer is achieved by defining a library of 2 neural networks shared across all tasks and scenarios. Training on each task should produce a task-specific source code solution (from scratch) and improve the overall usefulness of the shared networks. Below we outline the details of the specific models for each scenario along with baseline models.

#### 3.1 SHARED COMPONENTS

We refer to the 2 networks in the shared library as `net_0` and `net_1`. Both networks have similar architectures: they take a  $28 \times 28$  monochrome image as input and pass this sequentially through two fully connected layers each with 256 neurons and ReLU activations. The last hidden vector is passed through a fully connected layer and a softmax to produce a 10 dimensional output (`net_0`) or 4 dimensional output (`net_1`) to feed to the differentiable interpreter. Note that the output sizes are chosen to match the number of classes of MNIST digits and arithmetic operators respectively.

If we create an interpreter model containing  $N$  untrained networks, and part of the interpreter uses a parameter `net_choice = Param(N)` to deciding which network to apply, then the system effectively sees one large network, which cannot usefully be split apart into the  $N$  components after training. To avoid this, we enforce that no more than one untrained network is introduced at a time (i.e. the first task has access to only `net_0`, and all other tasks have access to both nets). We find that this breaks the symmetry sufficiently to learn separate, useful classifiers.

#### 3.2 ADD2X2 MODEL

For the ADD2X2 scenario we build a model capable of writing short straight line algorithms with up to 4 instructions. The model consists of a read head containing `net_0` and `net_1` (with the exception of the very first task, which only has access to `net_0`, as discussed above) which are connected to a set of registers each capable of holding integers in the range  $0, \dots, M$ , where  $M = 18$ . The head is initialized reading the top left cell of the  $2 \times 2$  grid, and at each step in the program, one instruction can be executed from the following instruction set:

- NOOP: a trivial no-operation instruction

<sup>1</sup>Note that for simplicity, our toy system ignores operator precedence and executes operations from left to right - i.e. the sequence in the text is executed as  $((d_1 \text{ op}_1 d_2) \text{ op}_2 d_3)$ .

<sup>2</sup>200 handwritten examples of each operator were collected from a single author to produce a training set of 600 symbols and a test set of 200 symbols from which to construct random  $2 \times 2$  grids.

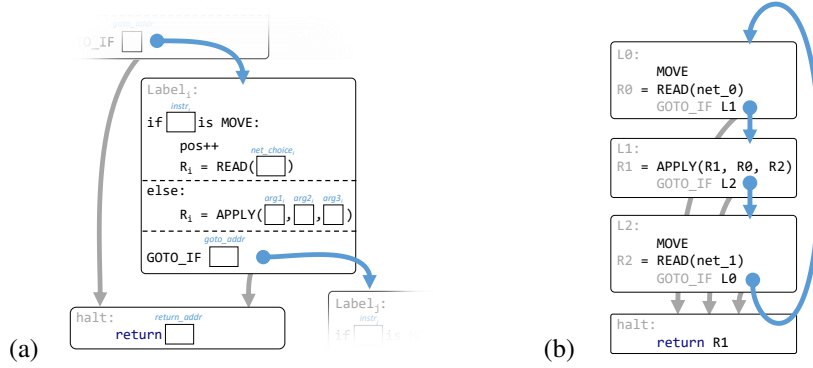


Figure 4: Overview of the MATH model. (a) The general form of a block in the model. Blue elements are learnable. (b) A loop-based solution to the task in the MATH scenario.

- `MOVE_NORTH`, `MOVE_EAST`, `MOVE_SOUTH`, `MOVE_WEST`: translate the head (if possible) and return the result of applying the neural network chosen by `net_choice` to the image in the new cell
- `ADD ( $\cdot, \cdot$ )`: accepts two register addresses and returns the sum of their contents.

where the parameter `net_choice` is to be learned and decides which of `net_0` and `net_1` to apply.

To construct each line of code requires choosing an instruction and (in the case of `SUM`) addresses of arguments for that instruction. We follow Feser et al. (2016) and allow each line to store its result in a separate immutable register. Finally, we learn a parameter specifying which register to return after execution of the program. An example program in this model is shown in Fig. 3(a). Even this simple model permits  $\sim 10^7$  syntactically distinct programs for the differentiable interpreter to search over.

**Baseline** To compare with a purely neural approach, we construct a baseline in which the interpreter is replaced with a task specific neural network. This model (which resembles a multitask network) contains two shared networks with the same architecture as `net_0` and `net_1` which are applied to the 4 digits to produce a 10 dimensional and a 4 dimensional embedding for each digit. We pass the concatenation of all embeddings to a task specific network consisting of a single ReLU hidden layer of 128 neurons. The task specific network outputs a classification of the sum into one of the 19 possible answers.

### 3.3 APPLY2X2 MODEL

We adapt the `ADD2X2` model to the `APPLY2X2` scenario by initializing three immutable registers with the auxiliary integers supplied with each  $2 \times 2$  operator grid [see Fig. 2(b)]. In addition, we swap the `ADD ( $\cdot, \cdot$ )` instruction for `APPLY ( $\cdot, \cdot, \cdot$ )`. The action of `APPLY (a, b, op)` is to interpret the integer stored at `op` as an arithmetic operator and to compute  $a \text{ op } b$ . All operations are performed modulo  $(M + 1)$  and division by zero returns  $M$ . In total, this model exposes a program space of size  $\sim 10^{12}$  syntactically distinct programs.

**Baseline** Additional task specific networks are added to the baseline from the `ADD2X2` model. These networks concatenate embeddings from the shared networks with one-hot representations of the auxiliary integers before passing through 3 hidden layers of 128 neurons each<sup>3</sup>.

### 3.4 MATH MODEL

We design the final scenario to investigate the synthesis of more complex control flow than straight line code. A natural solution to execute the expression on the tape is to build a loop with a body that alternates between moving the head and applying the operators [see Fig. 4(b)]. This loopy solution has the advantage that it generalises to handle arbitrary length arithmetic expressions.

<sup>3</sup>Using a validation data set, we find that a deeper task-specific network is needed to achieve good performance in the `APPLY2X2` tasks. Since the perceptual part of this task is very simple, we believe that the additional capacity is required to handle the nonlinear arithmetic operations.

Fig. 4(a) shows the basic architecture of the interpreter used in this scenario. We provide a set of blocks each containing the instruction `MOVE` or `APPLY`. A `MOVE` instruction increments the position of the head and loads the new symbol into a block specific immutable register using either `net_0` or `net_1` as determined by a block specific `net_choice`. After executing the instruction, the interpreter executes a `GOTO-IF` statement which checks whether the head is over the end of the tape and if not then it passes control to the block specified by `goto_addr`, otherwise control passes to a `halt` block which returns a chosen register value and exits the program. This model describes a space of  $\sim 10^6$  syntactically distinct programs.

**Baseline** To compare the generalisation properties of the proposed model with standard sequential neural architectures we add a recurrent neural network to the bank of task-specific networks in our baseline. At each step, this network takes in the shared embeddings of the current symbol, updates an LSTM hidden state and then proceeds to the next symbol. We make a classification of the final answer using the last hidden states of the LSTM. After tuning on a validation data set, we find that we achieve best performance with a 2 layer LSTM with 256 elements in each hidden state.

## 4 EXPERIMENTS

In this section we report results from three sets of experiments: We first demonstrate that learning with weak supervision is possible in the proposed settings by examining a single task, we then show that NTPT shows significant advantages over the baseline model when learning on a sequence of related tasks. Finally we demonstrate the generalization properties of learned NTPT models afforded by the source code representation.

### 4.1 WEAK SUPERVISION

Fig. 5 shows a comparison of the test accuracy of our weakly supervised NTPT model on the first `ADD2X2` task (summing up the top row) with two alternative models: (1) a strongly supervised model (a classifier (identical to `net_0`) directly trained on labelled digits) and (2) the purely neural baseline described above. In each case we report the accuracy for the task on a held-out test set containing 10k examples, and we also extract just the classifier component from each model and measure accuracy directly on a digit classification task<sup>4</sup>.

Each model is trained on 64k data instances drawn randomly from a data set containing either 4k distinct  $2 \times 2$  grid examples or 1k distinct examples.

We draw two conclusions from these experiments: (1) When the data set is sufficiently varied (4k examples), both of the weakly supervised models are able to complete the task as well as a model with strong supervision, and (2) By restricting the data set size to 1k examples, we can arrange a regime whereby weak supervision does not produce an accurate result on a single task, meaning that we must pool knowledge from many tasks to succeed in any one task. We work in this regime in the next section.

### 4.2 LIFELONG LEARNING

To test knowledge transfer between tasks we train on batches of data drawn from a probability distribution over all 8 tasks in the `ADD2X2` and `APPLY2X2` scenarios which evolves in time according to the schedule shown in the top left of Fig. 6(a). The identity of the task is provided with each batch such that the correct NTPT model or task-specific baseline network can be chosen. The learning rate for the shared networks and the task-specific components is tuned separately, and using an RMSProp

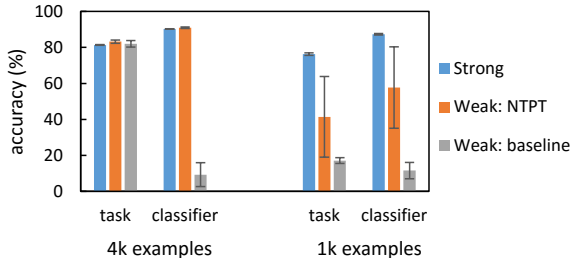


Figure 5: Performance of strongly and weakly supervised models on a single task in different data regimes. Error bars show standard deviation across 10 random restarts.

<sup>4</sup>Note that there is no reason for the digit embeddings to emit interpretable class labels in the purely neural baseline, and we include these values only for completeness



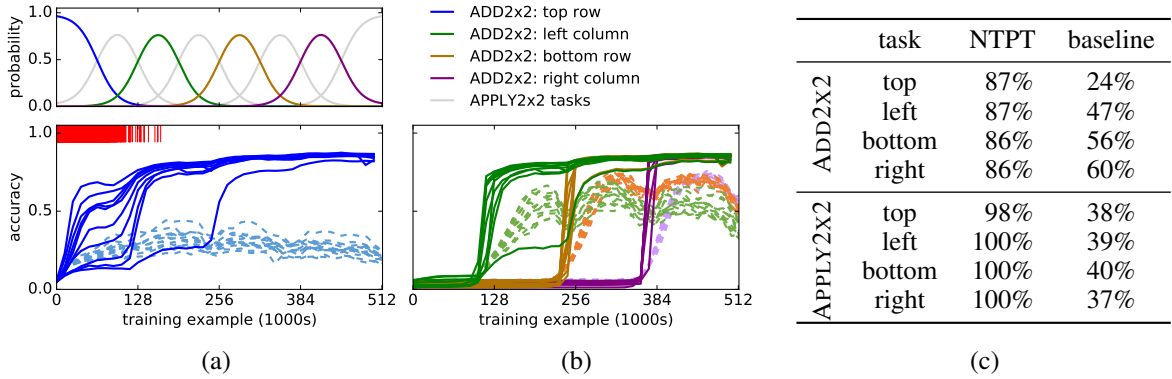


Figure 6: Lifelong learning with NTPT. (a) top: the sequential learning schedule for all 8 tasks, bottom: performance of NTPT (solid) and the baseline (dashed) on the first ADD2X2 task. (b) performance on the remaining ADD2X2 tasks. (c) Final accuracy on all tasks.

optimizer we find that for best knowledge transfer the learning rate of the shared networks should be 100 fold smaller than the task-specific components.

Fig. 6(a) shows the accuracy of NTPT and the baseline model on the first task of ADD2X2 (summing up the top row), for 10 random restarts. In all cases NTPT learns correct source code to solve the problem. Furthermore, we observe 3 key features:

- NTPT shows *reverse transfer*: even when we have stopped presenting examples for the first task (such examples are indicated by the red bars), the performance on this task continues to increase. We verify that this is due to continuous improvement of `net_0` by observing that the accuracy on the ADD2X2 task closely tracks measurements of the accuracy of `net_0` directly on the digit classification task (the final accuracy of `net_0` on the direct classification task is 93%).
- With the chosen balance of learning rates NTPT does not display catastrophic forgetting.
- NTPT considerably outperforms the baseline whose performance on ADD2X2 tasks starts to drop while training on later tasks.

Fig. 6(b) shows the performance on the remaining ADD2X2 tasks. We see that sharing of meaningful classifiers allows NTPT to learn solutions faster and with higher accuracy than the baseline. The results are similar for the APPLY2X2 scenario, with Fig. 6(c) showing that NTPT solves these tasks with  $\sim 100\%$  accuracy due to the simple nature of the operator data set, while the baseline struggles due to the complexity of the required program.

#### 4.3 GENERALIZATION

In the final experiment we take `net_0/1` from the end of the experiment above and start training on the MATH scenario with arithmetic expressions containing 2 digits. The loop structure of the MATH model introduces many local optima into the optimisation landscape and only 2/100 random restarts converge on a correct program. During a successful training run, the accuracy of `net_0` on the digit classification task increases from 93% to 95%, and the accuracy of `net_1` remains at 100%. Once the inferred source code is discretized, the model generalises well to longer expressions containing  $N$  digits, with the accuracy following the expected form  $0.95^N$  due to the repeated application of `net_0`. The LSTM baseline on the other hand shows excellent performance on  $N = 2$ , but does not generalize well (see Fig. 7)

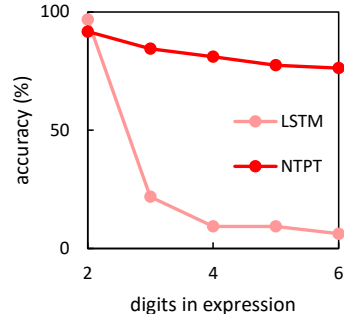


Figure 7: Generalisation behaviour on MATH expressions of varying length after training on 2 digit expressions.

## 5 RELATED WORK

**Lifelong Machine Learning.** We operate in the paradigm of Lifelong Machine Learning (LML) (Thrun, 1994; 1995; Thrun & O’Sullivan, 1996; Silver et al., 2013; Chen et al., 2015), where a

learner is presented a sequence of different tasks and the aim is to retain and re-use knowledge from earlier tasks to more efficiently and effectively learn new tasks. This is distinct from related paradigms of multitask learning (presentation of a finite set of tasks simultaneously rather than in sequence (Caruana, 1997; Kumar & Daume III, 2012; Luong et al., 2015; Rusu et al., 2016)), transfer learning (transfer of knowledge from a source to target domain without notion of knowledge retention (Pan & Yang, 2010)), and curriculum learning (training a single model for a single task of varying difficulty (Bengio et al., 2009)).

The challenge for LML with neural networks is the problem of catastrophic forgetting: if the distribution of examples changes during training, then neural networks are prone to forget knowledge gathered from early examples. Solutions to this problem involve instantiating a knowledge repository (KR) either directly storing data from earlier tasks or storing (sub)networks trained on the earlier tasks with their weights frozen. This knowledge base allows either (1) rehearsal on historical examples (Robins, 1995), (2) rehearsal on virtual examples generated by the frozen networks (Silver & Mercer, 2002; Silver & Poirier, 2006) or (3) creation of new networks containing frozen sub networks from the historical tasks (Rusu et al., 2016; Shultz & Rivest, 2001)

To frame our approach in these terms, our KR contains partially-trained neural network classifiers which we call from learned source code. Crucially, we never freeze the weights of the networks in the KR: all parts of the KR can be updated during the training of all tasks - this allows us to improve performance on earlier tasks by continuing training on later tasks (so-called reverse transfer). Reverse transfer has been demonstrated previously in systems which assume that each task can be solved by a model parameterized by an (uninterpretable) task-specific linear combination of shared basis weights (Ruvolo & Eaton, 2013). The representation of task-specific knowledge as source code, learning from weak supervision, and shared knowledge as a deep neural networks distinguishes this work from the linear model used in Ruvolo & Eaton (2013).

**Neural Networks Learning Algorithms.** Recently, extensions of neural networks with primitives such as memory and discrete computation units have been studied to learn algorithms from input-output data (Graves et al., 2014; Weston et al., 2014; Joulin & Mikolov, 2015; Grefenstette et al., 2015; Kurach et al., 2015; Kaiser & Sutskever, 2016; Reed & de Freitas, 2016; Bunel et al., 2016; Andrychowicz & Kurach, 2016; Zaremba et al., 2016; Graves et al., 2016; Riedel et al., 2016; Gaunt et al., 2016; Feser et al., 2016). Whereas many of these works use a neural network controller managing a differentiable computer architecture, we flip this relationship. The controller in our approach is a differentiable interpreter that is expressible as source code and can make calls to neural network components.

With the exception of Reed & de Freitas (2016) and (Graves et al., 2016), the methods above operate on inputs that are (arrays of) integers. In Reed & de Freitas (2016), this comes at the price of extremely strong supervision, where the learner is shown all intermediate steps to solving a problem; our learner only observes input and output examples. Reed & de Freitas (2016) also show the performance of their system in a multitask setting, however, in some cases additional tasks harm performance of the model and they freeze parts of their model when adding to their library of functions. Only Bunel et al. (2016), Riedel et al. (2016) and Gaunt et al. (2016) aim to consume and produce source code that can be provided by a human (e.g. as sketch of a solution) to or returned to a human (to potentially provide feedback).

## 6 DISCUSSION

We have presented NEURAL TERPRET, a framework for building end-to-end trainable models that structure their solution as a library of functions represented as source code or neural networks. Experimental results show that these models can successfully be trained in a lifelong learning context, and they are resistant to catastrophic forgetting; in fact, they show that even after instances of earlier tasks are no longer presented to the model, performance still continues to improve.

Learning neural network models within differentiable interpreters has several benefits. First, learning programs imposes a bias that favors learning models that exhibit strong generalization, as illustrated by many works on program-like neural networks. Second, the source code components are interpretable by humans, allowing incorporation of domain knowledge describing the shape of the problem through the source code structure. Third, source code components can be inspected, and



the neural network components can be queried with specific instances to inspect whether the shared classifiers have learned the expected mappings. A final benefit is that the differentiable interpreter can be seen as *focusing the supervision*. If a component is un-needed for a given task, then the differentiable interpreter can choose not to use the component, which shuts off any gradients from flowing to the component. We speculate that this could be a reason for the models being resistant to catastrophic forgetting, as the model either chooses to use a classifier (which further trains it as on related tasks and leads to reverse transfer), or ignores it (which leaves the component unchanged).

It is known that differentiable interpreters are difficult to train (Kurach et al., 2015; Neelakantan et al., 2016; Gaunt et al., 2016), and being dependent on differentiable interpreters is the primary limitation of this work. However, if progress can be made on more robust training of differentiable interpreters (perhaps extending ideas in Neelakantan et al. (2016); Feser et al. (2016)), then we believe there to be great promise in using the models we have presented here to build large lifelong neural networks.

## REFERENCES

- Martin Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.
- Marcin Andrychowicz and Karol Kurach. Learning efficient algorithms with hierarchical attentive memory. *arXiv preprint arXiv:1602.03218*, 2016.
- Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (ICML)*, pp. 41–48, 2009.
- Rudy Bunel, Alban Desmaison, Pushmeet Kohli, Philip H. S. Torr, and M. Pawan Kumar. Adaptive neural compilation. *CoRR*, abs/1605.07969, 2016. URL <http://arxiv.org/abs/1605.07969>.
- Rich Caruana. Multitask learning. *Machine Learning*, 28:41–75, 1997.
- Zhiyuan Chen, Nianzu Ma, and Bing Liu. Lifelong learning for sentiment classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 750–756, 2015.
- John K. Feser, Marc Brockschmidt, Alexander L. Gaunt, and Daniel Tarlow. Neural functional programming. 2016. Submitted to ICLR 2017.
- Alexander L. Gaunt, Marc Brockschmidt, Rishabh Singh, Nate Kushman, Pushmeet Kohli, Jonathan Taylor, and Daniel Tarlow. Terpret: A probabilistic programming language for program induction. *CoRR*, abs/1608.04428, 2016. URL <http://arxiv.org/abs/1608.04428>.
- Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *CoRR*, abs/1410.5401, 2014. URL <http://arxiv.org/abs/1410.5401>.
- Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 2016.
- Edward Grefenstette, Karl Moritz Hermann, Mustafa Suleyman, and Phil Blunsom. Learning to transduce with unbounded memory. In *Proceedings of the 28th Conference on Advances in Neural Information Processing Systems (NIPS)*, pp. 1828–1836, 2015.
- Armand Joulin and Tomas Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems 2, [NIPS Conference, Denver, Colorado, USA, November 27-30, 1989]*, pp. 190–198, 2015.
- Łukasz Kaiser and Ilya Sutskever. Neural GPUs learn algorithms. In *Proceedings of the 4th International Conference on Learning Representations (ICLR)*, 2016. URL <http://arxiv.org/abs/1511.08228>.

- Abhishek Kumar and Hal Daume III. Learning task grouping and overlap in multi-task learning. *arXiv preprint arXiv:1206.6417*, 2012.
- Karol Kurach, Marcin Andrychowicz, and Ilya Sutskever. Neural random-access machines. In *Proceedings of the 4th International Conference on Learning Representations 2016*, 2015. URL <http://arxiv.org/abs/1511.06392>.
- Minh-Thang Luong, Quoc V Le, Ilya Sutskever, Oriol Vinyals, and Lukasz Kaiser. Multi-task sequence to sequence learning. In *International Conference on Learning Representations (ICLR)*, 2015.
- Michael McCloskey and Neal J Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *Psychology of learning and motivation*, 24:109–165, 1989.
- Arvind Neelakantan, Quoc V. Le, and Ilya Sutskever. Neural programmer: Inducing latent programs with gradient descent. In *Proceedings of the 4th International Conference on Learning Representations 2016*, 2016.
- Sinno Jialin Pan and Qiang Yang. A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359, 2010.
- Emilio Parisotto, Lei Jimmy Ba, and Ruslan Salakhutdinov. Actor-mimic: Deep multitask and transfer reinforcement learning. In *International Conference on Learning Representations (ICLR)*, 2015. URL <http://arxiv.org/abs/1511.06342>.
- Roger Ratcliff. Connectionist models of recognition memory: constraints imposed by learning and forgetting functions. *Psychological review*, 97(2):285, 1990.
- Scott E. Reed and Nando de Freitas. Neural programmer-interpreters. 2016.
- Sebastian Riedel, Matko Bosnjak, and Tim Rocktäschel. Programming with a differentiable forth interpreter. *CoRR*, abs/1605.06640, 2016. URL <http://arxiv.org/abs/1605.06640>.
- Anthony Robins. Catastrophic forgetting, rehearsal and pseudorehearsal. *Connection Science*, 7(2): 123–146, 1995.
- Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016.
- Paul Ruvolo and Eric Eaton. Ella: An efficient lifelong learning algorithm. *ICML (1)*, 28:507–515, 2013.
- Thomas R Shultz and Francois Rivest. Knowledge-based cascade-correlation: Using knowledge to speed learning. *Connection Science*, 13(1):43–72, 2001.
- Daniel L Silver and Robert E Mercer. The task rehearsal method of life-long learning: Overcoming impoverished data. In *Conference of the Canadian Society for Computational Studies of Intelligence*, pp. 90–101. Springer, 2002.
- Daniel L Silver and Ryan Poirier. Machine life-long learning with csmtl networks. In *AAAI*, 2006.
- Daniel L Silver, Qiang Yang, and Lianghao Li. Lifelong machine learning systems: Beyond learning algorithms. In *AAAI Spring Symposium: Lifelong Machine Learning*, pp. 49–55, 2013.
- Sebastian Thrun. A lifelong learning perspective for mobile robot control. In *Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 23–30, 1994.
- Sebastian Thrun. Is learning the n-th thing any easier than learning the first? In *Advances in Neural Information Processing Systems 8 (NIPS)*, pp. 640–646, 1995.
- Sebastian Thrun and Joseph O’Sullivan. Discovering structure in multiple learning tasks: The TC algorithm. In *Machine Learning, Proceedings of the Thirteenth International Conference (ICML)*, pp. 489–497, 1996.

Jason Weston, Sumit Chopra, and Antoine Bordes. Memory networks. In *Proceedings of the 3rd International Conference on Learning Representations 2015*, 2014. URL <http://arxiv.org/abs/1410.3916>.

Wojciech Zaremba, Tomas Mikolov, Armand Joulin, and Rob Fergus. Learning simple algorithms from examples. In *Proceedings of the 33rd International Conference on Machine Learning, ICML 2016*, pp. 421–429, 2016.